APPENDIX

In the Specification:

Second paragraph on page 6

Referring now to FIG. 1, and more specifically, [the] a first stage in [the] a pipeline is [the] a world transform stage 105, in which the graphics engine converts the vertices and normals of the triangle from the real world object space, which may be different for each object in the scene, to the shared world space, which is space shared by all of the objects to be rendered in the entire scene. This transform consists of a matrix-vector multiplication for each vertex and each normal. In [the] a second stage of the pipeline, [the] a lighting stage 110, the graphics engine takes the triangle's color and surface normal(s) and computes the effect of one or more light sources. The result is a color at each vertex. At the next stage in the pipeline, [the] a view transform stage 115, the graphics engine converts the vertices from the world space to a camera space, with the viewer (or camera) at the center or origin and all vertices then mapped relative from that origin. Additionally, in the view transform stage 115, the graphics engine applies a matrix-vector multiplication to each vertex calculated for the camera space.

Third paragraph starting on page 6

As further shown in FIG. 1, the next stage in the pipeline is [the] a projection transform stage 120. At the projection transform stage 120, the graphics engine maps the vertices for the camera space to the actual view space. This includes the perspective transformation from 3D to 2D. Accordingly, at this point in the pipeline, the vertices are effectively two-dimensional to which perspective effects (i.e., depth foreshortening) have been applied. Accordingly, the third (z) coordinate is only needed to indicate the relative front-to-back ordering of the vertices when the objects are rendered or drawn within the view space. Like the other two transform stages in the pipeline, the projection transform stage requires the application of a matrix-vector multiplication per each vertex. In [the] a clipping stage 125, the graphics engine clips the triangles or primitives to [that the] fit within the view space. Accordingly, the triangles or primitives which lie entirely off the side of the screen or behind the viewer are removed. Meanwhile, triangles or primitives which are only partially out of bounds are trimmed. This generally requires splitting the

resulting polygon into additional multiple triangles or primitives and processing each one of these additional triangles or primitives separately. Finally, in [the] a rasterization stage 130, the graphics engine converts those triangles to be displayed within the view space into pixels and computes the color value to be displayed at each pixel. This includes visible-surface determination (dropping pixels which are obscured by a triangle closer to the viewer), texture mapping, and alpha blending (transparency effects).

Second paragraph on page 15.

Graphics primitives can have several source regions as well as a destination region; but, for simplicity, consider [the] a case where there is one of each. In this case and referring to FIG. 4c, a previously dispatched (and currently processing/executing) primitive *D* 402 will have a set of source pixel locations or a source operand surrounded by a bounding box and a set of destination pixel locations or a destination operand which is also surrounded[ing] by a bounding box. A candidate (or new primitive which has yet to be processed/executed) *P* 404 will also have a set of source pixel locations or a source operand surrounded by a bounding box and a set of destination pixel locations or a destination operand which is also surrounded[ing] by a bounding box. In a preferred embodiment, in order to determine whether or not the candidate primitive *P* depends on the previously dispatched primitive *D*, the function *depend*(*P*,*D*) is computed. Now, if S_P is the source region of *P* and D_P is the destination region of *P* and, furthermore, S_D and D_D are the source and destination regions of *D*, the dependency between the two can be determined by the following equation:

Third paragraph on page 23.

A preferred embodiment of [the] hardware used to compute dependencies is shown in FIG. 7. As shown in FIG. 7, the bounding box coordinates for [the] destination regions and [the] source regions of [the] a new primitive are driven onto vertical buses for each operand. In the example shown, the new primitive has a single destination region 701[,] and two source regions 702 and 703. Additionally, the bounding box coordinates for [the] destination regions which are stored in each valid reservation station are driven onto horizontal bus lines. At [the] intersections which correspond to potential

hazards, bounding box coordinate overlap comparators 704 implement the *depend* function described earlier. For example, a first comparator 704a may compare the destination region (i.e., destination bounding box) of a previously dispatched primitive with the destination region of the new primitive, and subsequently generate a first resultant bit. Similarly, a second comparator 704b may compare the source region (i.e., source bounding box) of the previously dispatched primitive with the destination regional of the new primitive, and generate a second resultant bit. Furthermore, a third comparator 704c may compare the destination region of the previously dispatched primitive with the source region of the new primitive, and generate a third resultant bit. Although only three comparators are used in this example, alternatively, a different number of comparators may be utilized. Subsequently, a logic OR gate 706 receives the first, second, and third resultant bits and performs a logic OR operation in order to determine whether any dependencies exist between the previously dispatched primitive and the new primitive. A dependence vector is thus calculated by computing whether or not the destination regions or source regions of the new primitive overlap with the destination regions of each (previously dispatched) primitive which is currently executing. Bit *k* in the dependence vector is set if the new primitive must wait for the primitive stored in destination reservation station *k* to complete, where *k* is the position in the destination reservation station where the conflicting primitives' destination region coordinates are stored [–] (i.e., destination reservation station 1, destination reservation station 2, etc.). This dependence vector is stored in the candidate buffer reserved for the new primitive.

At the same time, the issue unit is testing the existing issue candidates to see if any are ready to be issued. If any of the accelerators/rasterizers are available, then the issue unit tests all of the dependence vectors in the candidate buffers. If any valid candidate buffer contains a dependence vector of all zeros, then the primitive in the candidate buffer can be passed to the available graphics accelerator/rasterizer for processing on the next cycle. In a preferred embodiment, if more than one primitive in the candidate buffer has no dependency conflicts then that entry which corresponds with the earlier primitive or the primitive which has been in the candidate buffer the longest is the one which is selected for processing. At the next clock cycle, the primitive from the

candidate buffer is issued by transferring the primitive data and the tag to the available accelerator and clearing the valid bit in order to free that space up in the candidate buffer so a new primitive can be passed from the fetch unit to the issue unit.

Second paragraph on page 24.

As explained earlier, as the accelerators/rasterizers complete execution and processing of a primitive, the tag corresponding with that primitive is returned to the issue unit. The issue unit will then clear the valid bit for the entry in the destination reservation station which corresponds[ing] with that primitive to free up that space in the destination reservation station. The issue unit will also use the tag corresponding with the completed primitive to clear valid bits in the source reservation stations in order to make these spaces available for other primitives. Finally, the tag is decoded and the corresponding bit in any dependence vector[s] in the candidate buffer is cleared. This removes all information associated with the completed primitive from the reservation stations and clears any dependencies for pending primitives associated with that primitive. Once all of the dependence bits in a candidate's dependence vector have been cleared, that candidate is then said to be eligible and can then pass to a graphics accelerator/rasterizer on a subsequent clock cycle.

In the Claims:

1. A method for determining dependencies between a first graphics primitive and a second graphics primitive, the method comprising:

calculating a first bounding box for the first graphics primitive, the first bounding box surrounding at least one source operand for the first graphics primitive;

calculating a second bounding box for the second graphic primitive, the second bounding box surrounding a destination operand of the second graphic primitive; and

determining whether the first bounding box and the second bounding box overlap, wherein a dependency is detected if the [boundary] bounding boxes overlap.

4. The method of claim [3] 1, wherein a write after read dependency is detected if the second bounding box overlaps the first bounding box.

7. The method of claim [6] 1, wherein a [write] read after write dependency is detected if the [second] first bounding box overlaps the [first] second bounding box.

8. A method for determining whether a dependency exists between a first graphics primitive and a second graphics primitive, comprising:

comparing a set of destination pixel locations of [pixels to be drawn by] the first graphics primitive with a set of destination pixel locations of [pixels to be drawn by] the second graphics primitive; and

determining whether a dependency exists between the first and second graphics primitives as a function of the comparison of the destination pixel locations of the first and second graphic [pixels to be drawn by each] primitive.

9. The method of claim 8 wherein the step of comparing [is comprised of] further comprises:

calculating a first bounding box which surrounds the set of destination pixel locations of [pixels to be drawn by] the first graphics primitive;

calculating a second bounding box which surrounds the set of destination pixel locations of [pixels to be drawn by] the second graphics primitive; and

comparing the first bounding box with the second bounding box.

11. A method for determining whether a dependency exists between a first graphics primitive and a second graphics primitive, comprising:

comparing a set of destination pixel locations of [pixels to be drawn by] the first graphics primitive with at least one set of source pixel locations of [pixels to be used as source operands by] the second graphics primitive; and

determining whether a dependency exists between the first and second graphics primitives as a function of the comparison.

12. The method of claim 11 wherein the step of comparing [is comprised of] further comprises:

calculating a first bounding box which surrounds the set of destination pixel locations of [pixels to be drawn by] the first graphics primitive;

calculating a second bounding box [which surrounds] for each of the at least one set of source pixel locations of [pixels to be used as source operands and by] the second graphics primitive; and

determining whether there is [dependfirst] dependency if the first and second bounding boxes overlap.

14. An apparatus for detecting dependencies between a first graphics primitive and a second graphics primitive, comprising:

a destination reservation station for storing a destination bounding box location for the first graphics primitive;

a source reservation station for storing a source bounding box location for the first graphics primitive; and

a first comparator for comparing the destination bounding box location for the first graphics primitive with a bounding box location of [the location of pixels to be drawn by] the second graphics primitive and generating a first resultant bit.

15. The apparatus of claim 14 further comprising:

a second comparator for comparing the source bounding box location for the first graphics primitive with a destination bounding box location of [the location of pixels to be drawn by] the second graphics [primitve] primitive and generating a second resultant bit.

16. The apparatus of claim 15 further comprising:

a third comparator for comparing the destination bounding box location for the first graphics primitive with a source bounding box location of [the location of source pixels to be used by] the second graphics primitive and generating a third resultant bit.

22. The method of claim [3] 20 wherein the step of detecting is comprised of:

comparing the destination region bounding box coordinates for the primitives in the at least two which have not yet been completely processed with the destination region bounding box coordinates and the source region bounding box coordinates for the next primitive to be processed in order to detect a dependency; and

detecting a dependency if there is an overlap in the destination region bounding box coordinates for the primitives in the at least two which have not yet been completely processed and either the destination region bounding box coordinates or the source region bounding box coordinates for the next primitive to be processed.